2 The OAuth dance

This chapter covers

- An overview of the OAuth 2.0 protocol
- The different components in an OAuth 2.0 system
- How different components communicate with each other
- What different components communicate to each other

By now, you have a decent overview of what the OAuth 2.0 protocol is and why it's important. You also likely have an idea of how and where you might want to use the protocol. But what steps do you have to take to make an OAuth transaction? What do you end up with when you're done with an OAuth transaction? How does this design make OAuth secure?

2.1 Overview of the OAuth 2.0 protocol: getting and using tokens

OAuth is a complex security protocol, with different components sending pieces of information to each other in a precise balance akin to a technological dance. But fundamentally, there are two major steps to an OAuth transaction: issuing a token and using a token. The token represents the access that's been delegated to the client and it plays a central role in every part of OAuth 2.0. Whereas the details of each

step can vary based on several factors, the canonical OAuth transaction consists of the following sequence of events:

1. The Resource Owner indicates to the Client that they would like the Client to act on their behalf (for example, "Go load my photos from that service so I can print them").
2. The Client requests authorization from the Resource Owner at the Authorization Server.
3. The Resource Owner grants authorization to the Client.
4. The Client receives a Token from the Authorization Server.
5. The Client presents the Token to the Protected Resource.

Different deployments of the OAuth process can handle each of these steps in slightly different ways, often optimizing the process by collapsing several steps into a single action, but the core process remains essentially the same. Next, we'll look at the most canonical example of OAuth 2.0.

2.2 Following an OAuth 2.0 authorization grant in detail

Let's take a look at an OAuth authorization grant process in detail. We're going to be looking at all of the different steps between the different actors, tracing the HTTP requests and responses for each step. In particular, we'll be following the authorization code grant as used with a web-based client application. This client will be interactively authorized directly by the resource owner.

> **NOTE** The examples in this chapter are pulled from the exercise code that we'll be using later in this book. Although you don't need to understand the exercises to follow what's going on here, it might help to look at appendix A and run some of the completed examples to try this out. Also, note that the use of *localhost* throughout these examples is purely coincidental, as OAuth can and does work across multiple independent machines.

The *authorization code grant* uses a temporary credential, the authorization code, to represent the resource owner's delegation to the client, and it looks like what is shown in figure 2.1.

Let's break this down into individual steps. First, the resource owner goes to the client application and indicates to the client that they would like it to use a particular protected resource on their behalf. For instance, this is where the user would tell the printing service to use a specific photo storage service. This service is an API that the client knows how to process, and the client knows that it needs to use OAuth to do so.

How do I find the server?

To remain maximally flexible, OAuth pushes many details of a real API system out of scope. In particular, the way that the client knows how to talk to a given protected resource or how the client finds the authorization server tied to that protected

(continued)

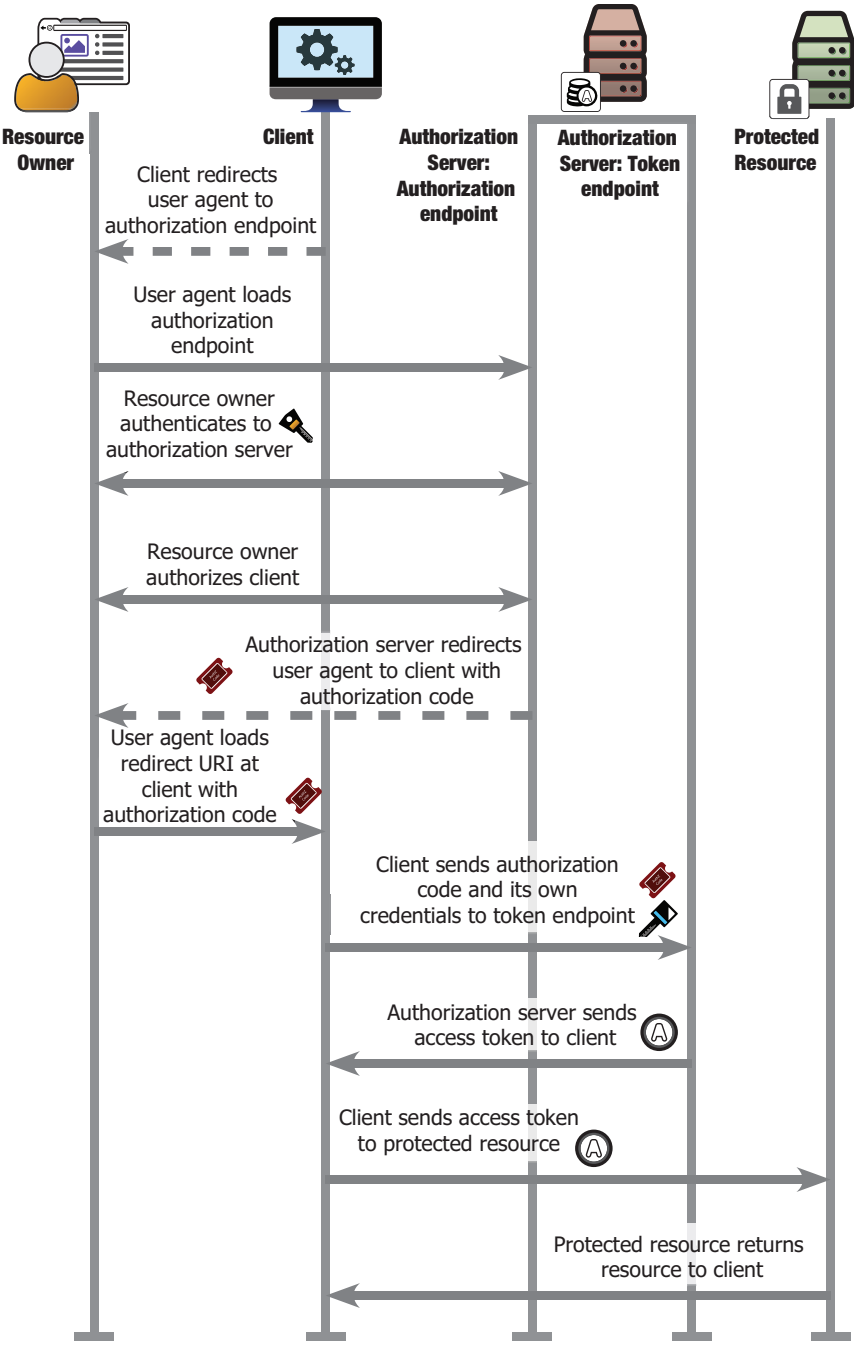

Figure 2.1 The authorization code grant in detail

resource isn't specified by OAuth. Some protocols built on top of OAuth, such as OpenID Connect and User Managed Access (UMA), do solve these problems in standard ways, and we'll cover those in chapters 13 and 14. For the purpose of demonstrating OAuth itself, we're assuming that the client has been statically configured to know how to talk to both the protected resource and the authorization server.

When the client realizes that it needs to get a new OAuth access token, it sends the resource owner to the authorization server with a request that indicates that the client is asking to be delegated some piece of authority by that resource owner (figure 2.2). For example, our photo printer could ask the photo-storage service for the ability to *read* the photos stored there.

Since we have a web client, this takes the form of an HTTP redirect to the authorization server's authorization endpoint. The response from the client application looks like this:

```
HTTP/1.1 302 Moved Temporarily
x-powered-by: Express
Location: http://localhost:9001/authorize?response_type=code&scope=foo&client
_id=oauth-client-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&
state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1
Vary: Accept
Content-Type: text/html; charset=utf-8
Content-Length: 444
Date: Fri, 31 Jul 2015 20:50:19 GMT
Connection: keep-alive
```

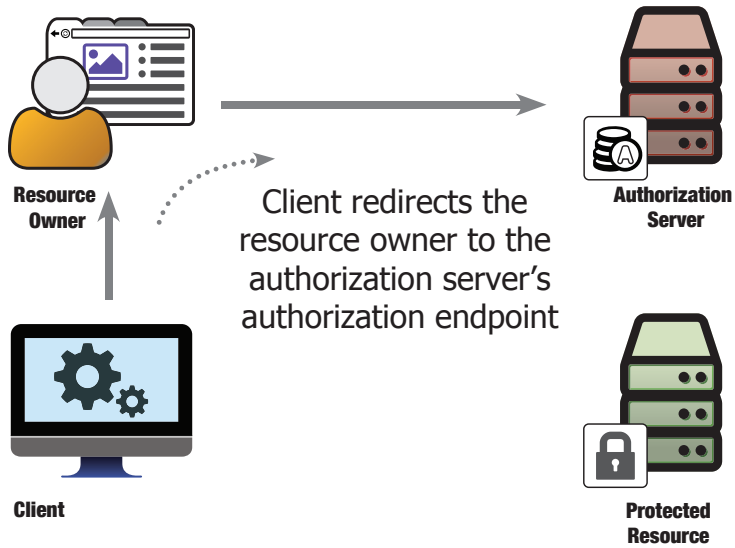

Figure 2.2 Sending the resource owner to the authorization server to start the process

This redirect to the browser causes the browser to send an HTTP GET to the authorization server.

```
GET /authorize?response_type=code&scope=foo&client_id=oauth-client
-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%
2Fcallback&state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1 HTTP/1.1
Host: localhost:9001
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost:9000/
Connection: keep-alive
```

The client identifies itself and requests particular items such as scopes by including query parameters in the URL it sends the user to. The authorization server can parse those parameters and act accordingly, even though the client isn't making the request directly.

Viewing the HTTP transaction

All of the HTTP transcripts were captured using off-the-shelf tools, and there are quite a number of them out there. Browser inspection tools, such as the Firebug plugin for Firefox, allow comprehensive monitoring and manipulation of front-channel communications. The back channel can be observed using a proxy system or a network packet capture program such as Wireshark or Fiddler.

Next, the authorization server will usually require the user to authenticate. This step is essential in determining who the resource owner is and what rights they're allowed to delegate to the client (see figure 2.3).

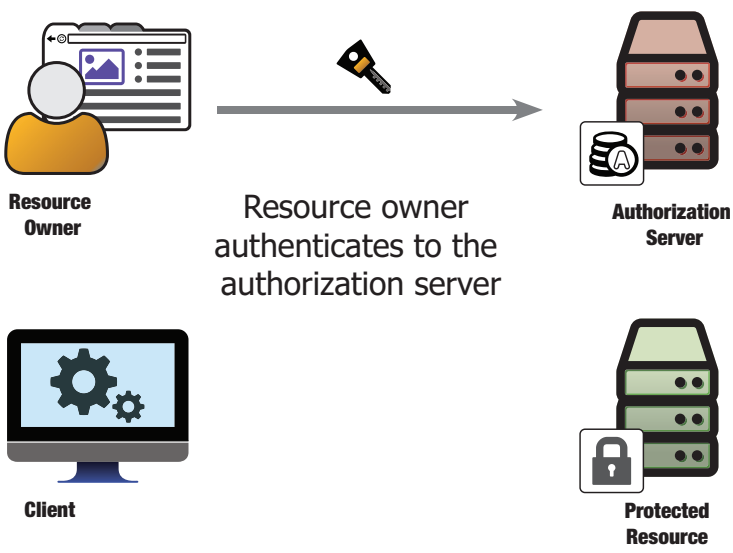

Figure 2.3 The resource owner logs in

The user's authentication passes directly between the user (and their browser) and the authorization server; it's never seen by the client application. This essential aspect protects the user from having to share their credentials with the client application, the antipattern that OAuth was invented to combat (as discussed in the last chapter).

Additionally, since the resource owner interacts with the authorization endpoint through a browser, their authentication happens through a browser as well. Thus, a wide variety of authentication techniques are available to the user authentication process. OAuth doesn't dictate the authentication technology, and the authorization server is free to choose methods such as a username/password pair, cryptographic certificates, security tokens, federated single-sign-on, or any number of other possibilities. We do have to trust the web browser to a certain extent here, especially if the resource owner is using a simple authentication method such as username and password, but the OAuth protocol is designed to protect against several major kinds of browser-based attacks, which we'll cover in chapters 7, 8, and 9.

This separated approach also insulates the client from changes to the user's authentication methods, allowing a simple client application to benefit from emergent techniques such as risk-based heuristic authentication applied at the authorization server. This doesn't convey any information about the authenticated user to the client, however; this is a topic we'll cover in depth in chapter 13.

Next, the user authorizes the client application (figure 2.4). In this step, the resource owner chooses to delegate some portion of their authority to the client application, and the authorization server has many different options to make this work. The client's request can include an indication of what kind of access it's looking for (known as the OAuth scope, discussed in section 2.4). The authorization server can allow the user to

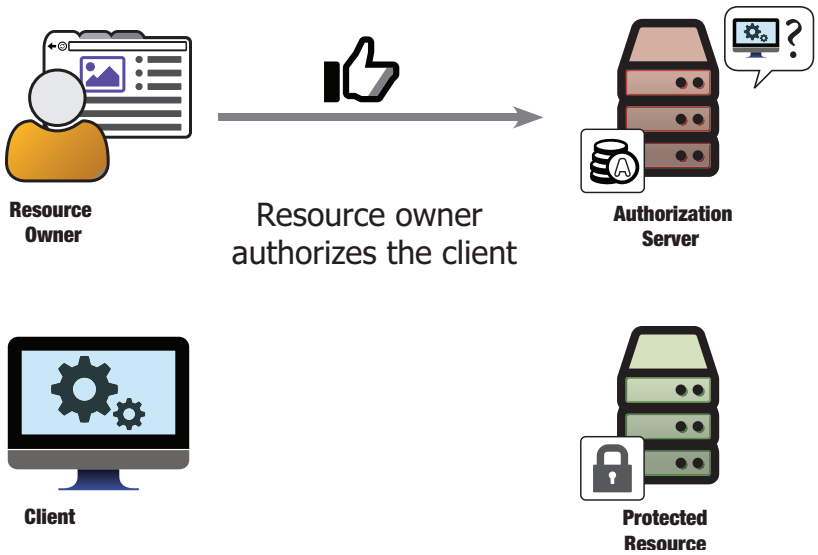

Figure 2.4 The resource owner approves the client's access request

deny some or all of these scopes, or it can let the user approve or deny the request as a whole.

Furthermore, many authorization servers allow the storage of this authorization decision for future use. If this is used, then future requests for the same access by the same client won't prompt the user interactively. The user will still be redirected to the authorization endpoint, and will still need to be logged in, but the decision to delegate authority to the client will have already been made during a previous attempt. The authorization server can even override the end user's decision based on an internal policy such as a client whitelist or blacklist.

Next, the authorization server redirects the user back to the client application (figure 2.5).

This takes the form of an HTTP redirect to the client's `redirect_uri`.

```
HTTP 302 Found
Location: http://localhost:9000/oauth_callback?code=8V1pr0rJ&state=Lwt50DDQKU
B8U7jtfLQCVGDL9cnmwHH1
```

This in turn causes the browser to issue the following request back to the client:

```
GET /callback?code=8V1pr0rJ&state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1 HTTP/1.1
Host: localhost:9000
```

Notice that this HTTP request is to the *client* and not on the *authorization server*.

```
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost:9001/authorize?response_type=code&scope=foo&client_
id=oauth-client-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&
state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1
Connection: keep-alive
```

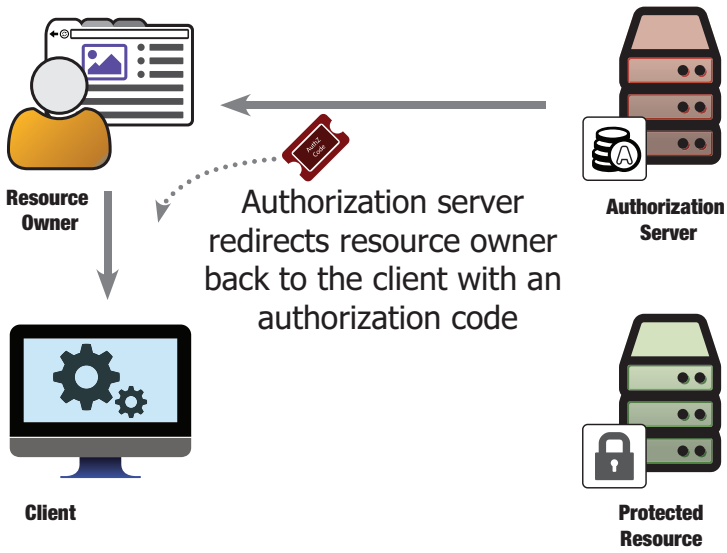

Figure 2.5 The authorization code is sent back to the client

Since we're using the *authorization code* grant type, this redirect includes the special `code` query parameter. The value of this parameter is a one-time-use credential known as the *authorization code*, and it represents the result of the user's authorization decision. The client can parse this parameter to get the authorization code value when the request comes in, and it will use that code in the next step. The client will also check that the value of the `state` parameter matches the value that it sent in the previous step.

Now that the client has the `code`, it can send it back to the authorization server on its token endpoint (figure 2.6).

The client performs an HTTP `POST` with its parameters as a form-encoded HTTP entity body, passing its `client_id` and `client_secret` as an HTTP Basic authorization header. This HTTP request is made directly between the client and the authorization server, without involving the browser or resource owner at all.

```
POST /token
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic b2F1dGgtY2xpZW50LTE6b2F1dGgtY2xpZW50LXNlY3JldC0x

grant_type=authorization_code&
redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&code=8V1pr0rJ
```

This separation between different HTTP connections ensures that the client can authenticate itself directly without other components being able to see or manipulate the token request.

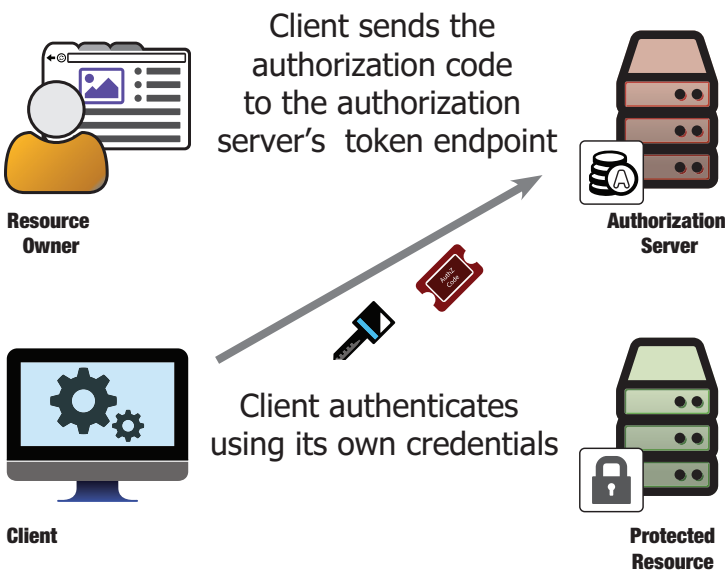

Figure 2.6 The client sends the code and its own credentials back to the authorization server

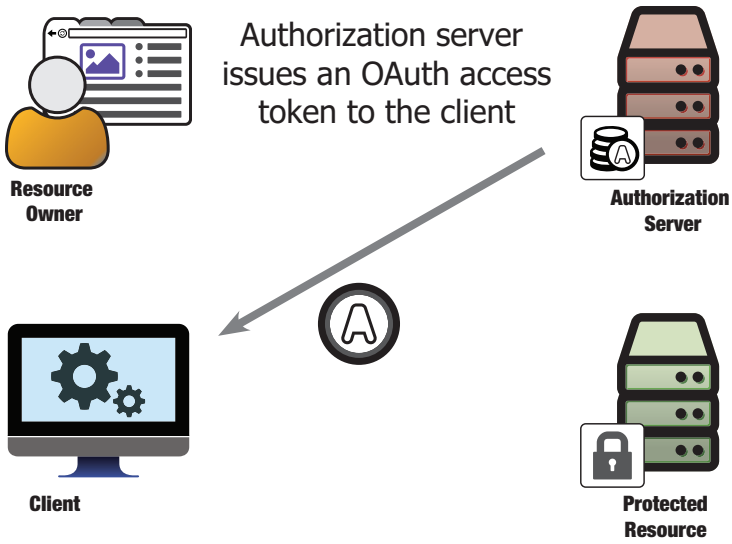

Figure 2.7 The client receives an access token

The authorization server takes in this request and, if valid, issues a token (figure 2.7). The authorization server performs a number of steps to ensure the request is legitimate. First, it validates the client's credentials (passed in the `Authorization` header here) to determine which client is requesting access. Then, it reads the value of the `code` parameter from the body and looks up any information it has about that authorization code, including which client made the initial authorization request, which user authorized it, and what it was authorized for. If the authorization code is valid, has not been used previously, and the client making this request is the same as the client that made the original request, the authorization server generates and returns a new access token for the client.

This token is returned in the HTTP response as a JSON object.

```
HTTP 200 OK
Date: Fri, 31 Jul 2015 21:19:03 GMT
Content-type: application/json

{
   "access_token": "987tghjkiu6trfghjuytrghj",
   "token_type": "Bearer"
}
```

The client can now parse the token response and get the access token value from it to be used at the protected resource. In this case, we have an OAuth Bearer token, as indicated by the `token_type` field in the response. The response can also include a refresh token (used to get new access tokens without asking for authorization again) as well as additional information about the access token, like a hint about the token's scopes and expiration time. The client can store this access token in a secure place for as long as it wants to use the token, even after the user has left.

The right to bear tokens

The core OAuth specifications deal with *bearer* tokens, which means that anyone who carries the token has the right to use it. All of our examples throughout the book will use bearer tokens, except where specifically noted. Bearer tokens have particular security properties, which are enumerated in chapter 10, and we'll take a look ahead at nonbearer tokens in chapter 15.

With the token in hand, the client can present the token to the protected resource (see figure 2.8).

The client has several methods for presenting the access token, and in this example we're going to use the recommended method of using the Authorization header.

```
GET /resource HTTP/1.1
Host: localhost:9002
Accept: application/json
Connection: keep-alive
Authorization: Bearer 987tghjkiu6trfghjuytrghj
```

The protected resource can then parse the token out of the header, determine whether it's still valid, look up information regarding who authorized it and what it was authorized for, and return the response accordingly. A protected resource has a number of options for doing this token lookup, which we'll cover in greater depth in chapter 11. The simplest option is for the resource server and the authorization server to share a database that contains the token information. The authorization server writes new tokens into the store when they're generated, and the resource server reads tokens from the store when they're presented.

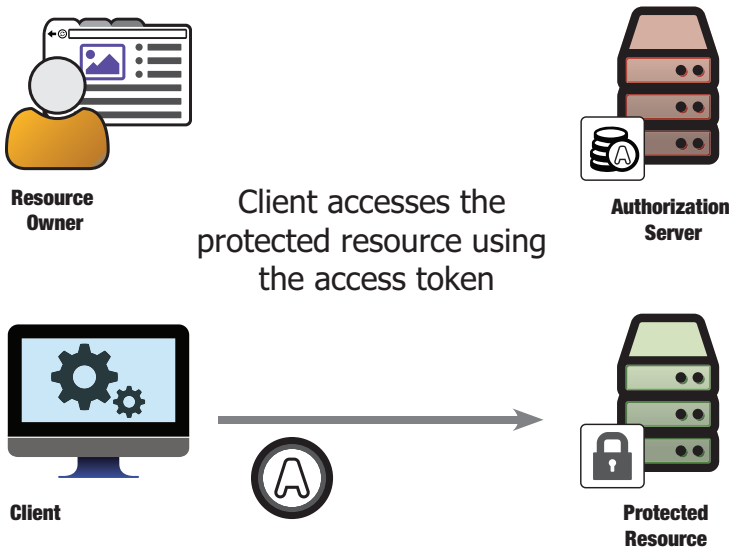

Figure 2.8 The client uses the access token to do things

2.3 OAuth's actors: clients, authorization servers, resource owners, and protected resources

As we touched on in the last section, there are four main actors in an OAuth system: clients, resource owners, authorization servers, and protected resources (figure 2.9). Each of these components is responsible for different parts of the OAuth protocol, and all work together to make the OAuth protocol work.

An OAuth *client* is a piece of software that attempts to access the protected resource on behalf of the resource owner, and it uses OAuth to obtain that access. Thanks to the design of the OAuth protocol, the client is generally the simplest component in an OAuth system, and its responsibilities are largely centered on obtaining tokens from the authorization server and using tokens at the protected resource. The client doesn't have to understand the token, nor should it ever need to inspect the token's contents. Instead, the client uses the token as an opaque string. An OAuth client can be a web application, a native application, or even an in-browser JavaScript application, and we'll cover the differences between these kinds of clients in chapter 6. In our cloud-printing example, the printing service is the OAuth client.

An OAuth *protected resource* is available through an HTTP server and it requires an OAuth token to be accessed. The protected resource needs to validate the tokens presented to it and determine whether and how to serve requests. In an OAuth architecture, the protected resource has the final say as to whether or not to honor a token. In our cloud-printing example, the photo-storage site is the protected resource.

A *resource owner* is the entity that has the authority to delegate access to the client. Unlike other parts of the OAuth system, the resource owner isn't a piece of software. In most cases, the resource owner is the person using the client software to access

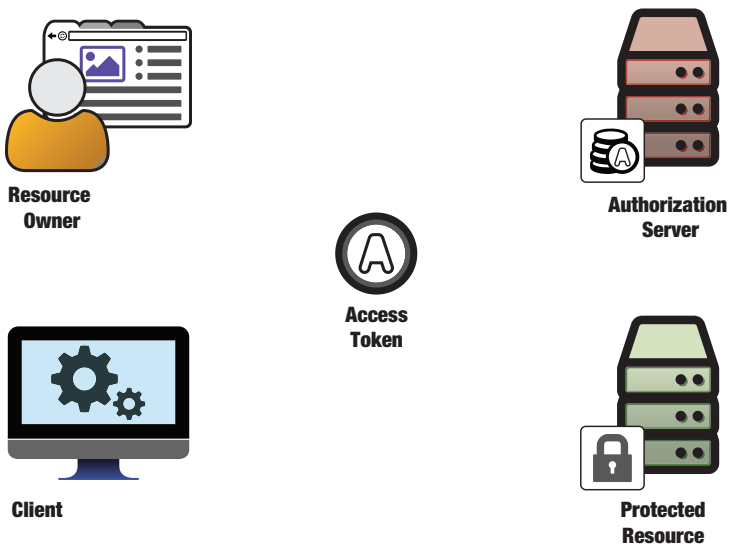

Figure 2.9 Major components of the OAuth 2.0 protocol

something they control. For at least part of the process, the resource owner interacts with the authorization server using a web browser (more generally known as the user agent). The resource owner might also interact with the client using a web browser, as they do in our demonstration here, but that's entirely dependent on the nature of the client. In our cloud-printing example, the resource owner is the end user who wants to print their photos.

An OAuth *authorization server* is an HTTP server that acts as the central component to an OAuth system. The authorization server authenticates the resource owner and client, provides mechanisms for allowing resource owners to authorize clients, and issues tokens to the client. Some authorization servers also provide additional capabilities such as token introspection and remembering authorization decisions. In our cloud-printing example, the photo-storage site runs its own in-house authorization server for its protected resources.

2.4 OAuth's components: tokens, scopes, and authorization grants

In addition to these actors, the OAuth ecosystem depends on several other mechanisms, both conceptual and physical. These are the bits that connect the actors in the previous section in a larger protocol.

2.4.1 Access tokens

An OAuth *access token*, sometimes known as just a *token* in casual reference, is an artifact issued by the authorization server to a client that indicates the rights that the client has been delegated. OAuth does not define a format or content for the token itself, but it always represents the combination of the client's requested access, the resource owner that authorized the client, and the rights conferred during that authorization (usually including some indication of the protected resource).

OAuth tokens are opaque to the client, which means that the client has no need (and often no ability) to look at the token itself. The client's job is to carry the token, requesting it from the authorization server and presenting it to the protected resource. The token isn't opaque to everyone in the system: the authorization server's job is to issue the token, and the protected resource's job is to validate the token. As such, they both need to be able to understand the token itself and what it stands for. However, the client is completely oblivious to all of this. This approach allows the client to be much simpler than it would otherwise need to be, as well as giving the authorization server and protected resource incredible flexibility in how these tokens are deployed.

2.4.2 Scopes

An OAuth *scope* is a representation of a set of rights at a protected resource. Scopes are represented by strings in the OAuth protocol, and they can be combined into a set by using a space-separated list. As such, the scope value can't contain the space character. The format and structure of the scope value are otherwise undefined by OAuth.

Scopes are an important mechanism for limiting the access granted to a client. Scopes are defined by the protected resource, based on the API that it's offering. Clients can request certain scopes, and the authorization server can allow the resource owner to grant or deny particular scopes to a given client during its request. Scopes are generally additive in nature.

Let's return to our cloud-printing example. The photo-storage service's API defines several different scopes for accessing the photos: read-photo, read-metadata, update-photo, update-metadata, create, and delete. The photo-printing service needs to be able only to read the photos in order to do its job, and so it asks for the `read-photo` scope. Once it has an access token with this scope, the printer is able to read photos and print images out as requested. If the user decides to use an advanced function that prints a series of photographs into a book based on their date, the printing service will need the additional `read-metadata` scope. Since this is an additional access, the printing service needs to ask the user to authorize them for this additional scope using the regular OAuth process. Once the printing service has an access token with both scopes, it can perform actions that require either of them, or both of them together, using the same access token.

2.4.3 Refresh tokens

An OAuth *refresh token* is similar in concept to the access token, in that it's issued to the client by the authorization server and the client doesn't know or care what's inside the token. What's different, though, is that the token is never sent to the protected resource. Instead, the client uses the refresh token to request new access tokens without involving the resource owner (figure 2.10).

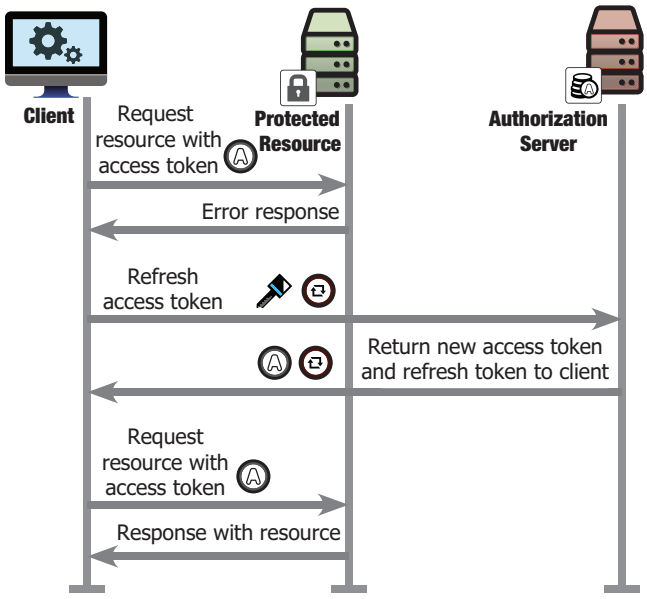

Figure 2.10 Using a refresh token

Why would a client need to bother with a refresh token? In OAuth, an access token could stop working for a client at any point. The user could have revoked the token, the token could have expired, or some other system trigger made the token invalid. The client will usually find out about the token being invalid by using it and receiving an error response. Of course, the client could have the resource owner authorize it again, but what if the resource owner's no longer there?

In OAuth 1.0, the client had no recourse but to wait for the resource owner's return. To avoid this, tokens in OAuth 1.0 tended to live forever until explicitly revoked. This is a bit problematic as it increases the attack surface for a stolen token: the attacker can keep using the stolen token forever. In OAuth 2.0, access tokens were given the option to expire automatically, but we still need a way to access resources when the user was no longer there. The refresh token now takes the place of the long-lived token, but instead of it being used to obtain resources, it's used only to get new access tokens that, in turn, can get the resources. This limits the exposure of the refresh token and the access token in separate but complementary ways.

Refresh tokens also give the client the ability to down-scope its access. If a client is granted scopes `A`, `B`, and `C`, but it knows that it needs only scope `A` to make a particular call, it can use the refresh token to request an access token for only scope `A`. This lets a smart client follow the security principle of least privilege without burdening less-smart clients with trying to figure out what privileges an API needs. Years of deployment experience have shown that OAuth clients tend to be anything but smart, but it's still good to have the advanced capability there for those that want to exercise it.

What then if the refresh token itself doesn't work? The client can always bother the resource owner again, when they're available. In other words, the fallback state for an OAuth client is to do OAuth again.

2.4.4 Authorization grants

An *authorization grant* is the means by which an OAuth client is given access to a protected resource using the OAuth protocol, and if successful it ultimately results in the client getting a token. This is likely one of the most confusing terms in OAuth 2.0, because the term is used to define both the specific mechanism by which the user delegates authority as well as the act of delegation itself. This is further confused by the authorization code grant type, which we laid out in detail previously, as developers will sometimes look at the authorization code that is passed back to the client and mistakenly assume that this artifact—and this artifact alone—is the authorization grant. Although it's true that the authorization code represents a user's authorization decision, it is not itself an authorization grant. Instead, the entire OAuth process is the authorization grant: the client sending the user to the authorization endpoint, then receiving the code, then finally trading the code for the token.

In other words, the authorization grant is the method for getting a token. In this book, as in the OAuth community as a whole, we'll occasionally refer to this as a *flow* of the OAuth protocol. Several different kinds of authorization grants exist in OAuth,

each with its own characteristics. We'll be covering these in detail in chapter 6, but most of our examples and exercises, such as those in the previous section, use the authorization code authorization grant type.

2.5 Interactions between OAuth's actors and components: back channel, front channel, and endpoints

Now that we know the different parts of an OAuth system, let's take a look at how exactly they communicate with each other. OAuth is an HTTP-based protocol, but unlike most HTTP-based protocols, OAuth communication doesn't always happen through a simple HTTP request and response.

> **OAuth over non-HTTP channels**
>
> Although OAuth is defined only in terms of HTTP, several specifications have defined how to move different parts of the OAuth process to non-HTTP protocols. For instance, draft standards exist that define how to use OAuth tokens over Generic Security Services Application Program Interface (GSS-API)[1] and Constrained Application Protocol (CoAP).[2] These can still use HTTP to bootstrap the process, and they tend to translate the HTTP-based OAuth components as directly as possible to these other protocols.

2.5.1 Back-channel communication

Many parts of the OAuth process use a normal HTTP request and response format to communicate to each other. Since these requests generally occur outside the purview of the resource owner and user agent, they are collectively referred to as back-channel communication (figure 2.11).

These requests and responses make use of all the regular HTTP mechanisms to communicate: headers, query parameters, methods, and entity bodies can all contain information vital to the transaction. Note that this might be a bit more of the HTTP stack than you're used to, as many simple web APIs allow the client developer to pay attention to the response body.

The authorization server provides a token endpoint that the client uses to request access tokens and refresh tokens. The client calls this endpoint directly, presenting a form-encoded set of parameters that the authorization server parses and processes. The authorization server then responds with a JSON object representing the token.

Additionally, when the client connects to the protected resource, it's also making a direct HTTP call in the back channel. The details of this call are entirely dependent on the protected resource, as OAuth can be used to protect an extraordinarily wide variety of APIs and architecture styles. In all of these, the client presents the OAuth

[1] RFC 7628 https://tools.ietf.org/html/rfc7628
[2] https://tools.ietf.org/html/draft-ietf-ace-oauth-authz

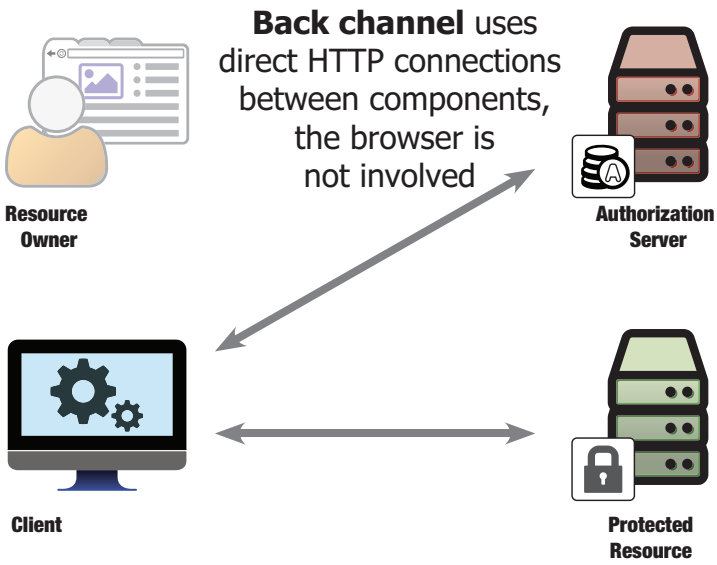

Figure 2.11 Back-channel communication

token and the protected resource must be able to understand the token and the rights that it represents.

2.5.2 *Front-channel communication*

In normal HTTP communication, as we saw in the previous section, the HTTP client sends a request that contains headers, query parameters, an entity body, and other pieces of information directly to a server. The server can then look at those pieces of information and figure out how to respond to the request, using an HTTP response containing headers, an entity body, and other pieces of information. However, in OAuth there are several instances in which two components cannot make direct requests of and responses to each other, such as when the client interacts with the authorization endpoint of the authorization server. Front-channel communication is a method of using HTTP requests to communicate indirectly between two systems through an intermediary web browser (figure 2.12).

This technique isolates the sessions on either side of the browser, which allows it to work across different security domains. For instance, if the user needs to authenticate to one of the components, they can do so without exposing their credentials to the other system. We can keep information separate and still communicate in the presence of the user.

How can two pieces of software communicate without ever talking to each other? Front-channel communication works by attaching parameters to a URL and indicating that the browser should follow that URL. The receiving party can then parse the incoming URL, as fetched by the browser, and consume the presented information. The receiving party can then respond by redirecting the browser back to a URL hosted by the originator, using the same method of adding parameters. The two parties are

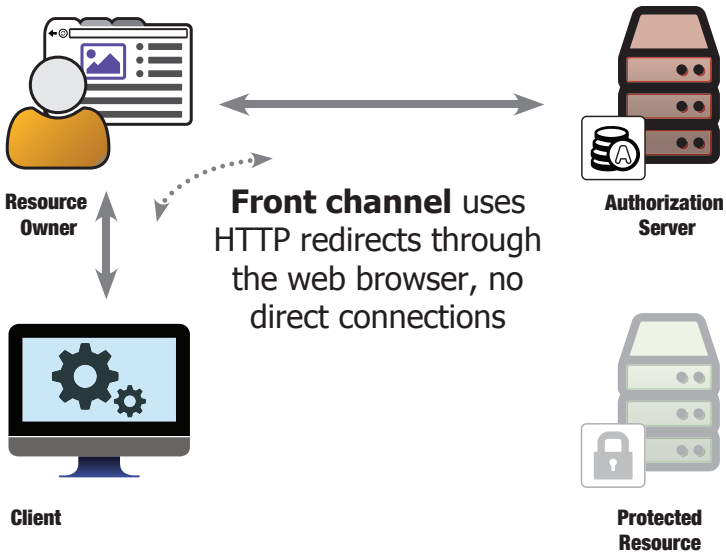

Figure 2.12 Front-channel communication

thus communicating with each other indirectly through the use of the web browser as an intermediary. This means that each front-channel request and response is actually a pair of HTTP request and response transactions (figure 2.13).

For example, in the authorization code grant that we saw previously, the client needs to send the user to the authorization endpoint, but it also needs to communicate certain parts of its request to the authorization server. To do this, the client sends an HTTP redirect to the browser. The target of this redirect is the server's URL with certain fields attached to it as query parameters:

```
HTTP 302 Found
Location: http://localhost:9001/authorize?client_id=oauth-client-1&response_
type=code&state=843hi43824h42tj
```

The authorization server can parse the incoming URL, just like any other HTTP request, and find the information sent from the client in these parameters. The authorization server can interact with the resource owner at this stage, authenticating them and asking for authorization over a series of HTTP transactions with the browser. When it's time to return the authorization code to the client, the authorization server sends an HTTP redirect to the browser as well, but this time with the client's `redirect_uri` as the base. The authorization server also includes its own query parameters in the redirect:

```
HTTP 302 Found
Location: http://localhost:9000/oauth_callback?code=23ASKBWe4&state=843hi438
24h42tj
```

When the browser follows this redirect, it will be served by the client application, in this case through an HTTP request. The client can parse the URL parameters from the

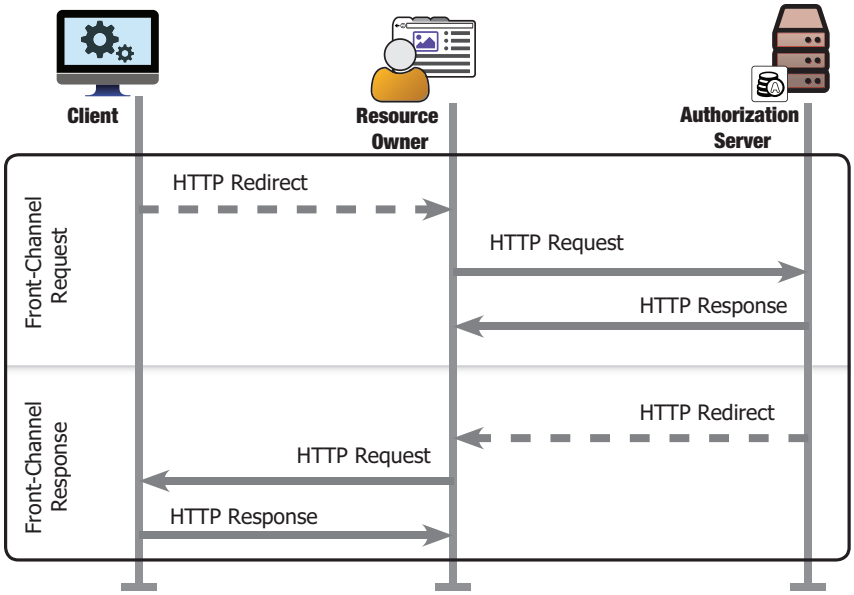

Figure 2.13 Parts of a front-channel request and response

incoming request. In this way, the client and authorization server can pass messages back and forth to each other through an intermediary without ever talking to each other directly.

What if my client isn't a web application?

OAuth can be used by both web applications and native applications, but both need to use the same front-channel mechanism to receive information back from the authorization endpoint. The front channel always uses a web browser and HTTP redirects, but they don't always have to be served by a regular web server in the end. Fortunately, there are a few useful tricks, such as internal web servers, application-specific URI schemes, and push notifications from a back-end service that can be used. As long as the browser can invoke a call on that URI, it will work. We'll explore all of these options in detail in chapter 6.

All information passed through the front channel is accessible to the browser, both to be read and potentially manipulated before the ultimate request is made. The OAuth protocol accounts for this by limiting the kinds of information that are passed through the front channel, and by making sure that none of the pieces of information used in the front channel can be used on their own to accomplish the task of delegation. In the canonical case we saw in this chapter, the authorization code can't be used by the browser directly, but instead it must be presented alongside the client's credentials in

the back channel. Some protocols, such as OpenID Connect, offer increased security through mechanisms for these front-channel messages to be signed by the client or authorization server to add a further layer of protection, and we'll look at that briefly in chapter 13.

2.6 Summary

OAuth is a protocol with many moving pieces, but it's built of simple actions that add up to a secure method for authorization delegation.

- OAuth is about *getting tokens* and *using tokens*.
- Different components in the OAuth system care about different parts of the process.
- Components use direct (back channel) and indirect (front channel) HTTP to communicate with each other.

Now that you've learned what OAuth is and how it works, let's start building things! In the next chapter, we'll build an OAuth client from scratch.